6D Object Pose Estimation Binaries

November 5, 2014

All data regarding our ECCV 14 paper can be downloaded from our project page: `http://www.inf.tu-dresden.de/index.php?node_id=3629&ln=en`. If you run into problems contact: eric <*dot*> brachmann <*at*> tu-dresden.de.

1 Overview

We provide Linux[1] executables for 6D pose estimation on RGB-D images. The implementation adheres to our ECCV 14 paper[1]. See the paper for details on the underlying algorithm and the evaluation procedure. You can use the binaries to reproduce the results reported in our paper, or to test our algorithm on your own data. This documentation describes the layout of training and test data assumed by the executables. We list command line arguments to set certain parameters, we describe the output generated, and give an example on how to use the binaries on our own 20 objects dataset.

We provide the following elements:

train_trees Executable that trains a random forest from the training set.

test_pose_estimation Executable that loads a trained forest and performs pose estimation on the test set.

create_split_full.py A python script used to split our 20 objects dataset into training and test sets.

create_split_small.py A python script used to split data of 5 of our 20 objects into training and test sets. It is used in the example at the end of this document.

create_split_util.py Utility functions for the python script mentioned above.

render_lib Rendering library used by `test_pose_estimation`.

shaders Shaders needed by the rendering library.

In order to run the executables OpenCV 2.4.2 and CUDA 5.5 has to be installed on your system.

[1] Compiled under Ubuntu 12.04. 64bit

2 Data

The executables rely on the following layout of the working directory:

```
training
    object1_data_folder
    object2_data_folder
    ...
test
    object1_data_folder
    object2_data_folder
    ...
background
    bg_data_folder
obj1.obj
obj2.obj
...
pc1.xyz
pc2.xyz
...
```

The working directory contains 3 folders: `test`, `training` and `background`. The first two folders contain one data folder per object. Each data folder has the following sub-folders: `rgb_noseg`, `depth_noseg`, `mask` and `info`. You find more information about these sub-folders and the files they contain in the documentation of our 20 objects dataset. The alphabetical order of the object folders determine the objects IDs. Hence, it is vital that folder names in `training` and `test` are mirrored. The folder `background` contains one data folder with the sub-folders `rgb_noseg`, `depth_noseg`. These images are used as negative class during the training of the random forest.

The working directory also contains one 3D mesh file (`objX.obj`) and one point cloud file (`pcX.xyz`) per object. The number `X` of these files is determined by the alphabetical order of the object folders within `training/test`. You can find more information on these files in our 20 objects dataset.

3 Training

In order to estimate poses on your test images you have to train a random forest first. To do so, you execute `train_trees` in your working directory. The program will access `training` and `background`. Table 1 lists possible command line arguments.

3.1 Output

The executable stores a binary random forest file (*.rf) in the working directory. The file name encodes some of the training parameters.

4 Test

Execute `test_pose_estimation` in your working directory. You specify an object to test via command line argument, and the program cycles through all test images of that object and compares the estimated pose with ground truth. After each image, the program will show you some qualitative results (in multiple windows). These include the estimated pose (blue bounding box), the ground truth pose (green bounding box), the probability map for this object, the object coordinate prediction of one tree, and the object coordinate ground truth. You continue with the next image by pressing any button. You can also switch to batch mode via command line argument. After processing all images, it will give you the ratio of correctly estimated poses. The program assumes that a random forest has been trained before. It will derive the file name from the parameters you specify. If you changed any parameters for training, you have to specify the same parameters for `test_pose_estimation`. Table 2 lists additional parameters that are specific to the test phase.

4.1 Output

The executable `test_pose_estimation` produces multiple output files that enable you to access the estimated poses or apply your own evaluation metric. After each completed run of the program a file `objX_bg1.txt` is created where `X` is the object ID. This file contains one line per test run. Each line is composed of 6 numbers:

1 Test object: ID of the object that has been tested.

2 Rotation object: 1/0 depending on whether the object was specified to be rotational symmetric or not.

3 Test images: Number of images processed.

4 Estimated poses: Ratio of images where the pose was estimated correctly (in percent).

5 Avg. forest time: Average time the forest evaluation took per image (in ms).

6 Avg. RANSAC time: Average time the energy minimization took per image (in ms).

The program will also create a new folder in the working directory, that has the same name as the random forest used. This folder contains one *.txt file per object (indicated by the ID after `tO` in the file name). Every time an images is processed by `test_pose_estimation` one new line is appended to the file of the specified object. Each line consists of 25 numbers:

1 Test image number.

2 Distance of the test image to the closest training image. This can be seen as an indication of difficulty.

3 Pose error between estimation and ground truth. Details about this measure can be found in the supplement of our ECCV 14 paper[1].

4-12 Rotation matrix of the estimated pose in row first order. Our definition of object pose can be found in the documentation of our 20 object dataset.

13-15 Translation vector (xyz) of the estimated pose (in meters).

16 Energy value associated with the estimated pose. It can be used for evaluating the detection performance of the system.

17-20 Estimated 2D bounding box of the object to detect. The 4 numbers are: x and y coordinates of the top left corner of the bounding box (in px), and width and height (also in px).

21-24 Ground truth 2D bounding box.

25 Indication (0/1) whether the estimated bounding box overlaps sufficiently with the ground truth bounding box (70% intersection over union). This can be used for evaluating the detection performance of the system.

5 Example

In order to follow this example you need to download our 20 object dataset (`20objects.zip`) and our background set (`bgs.zip`). Copy the contents of those 2 archives in a folder named `data`. This folder should then have 62 subfolders. Copy all contents of the executable archive (`eccv14bin.zip`) into the same folder `data` resides in. This should leave you with the following file layout:

```
<your_base_folder>
    data
        BG_Rooms
        BG_Rooms_Objs
        Kinfu_Audiobox1_dark
        ...
    render_lib
    shaders
    create_split_small.py
    create_split_full.py
    create_split_util.py
    test_pose_estimation
    train_trees
```

Execute `create_split.py` by typing:

```
> python create_split.py
```

The script will create a new working directory (`split_5objects...`) with `background`, `training` and `test` folders linking to the appropriate images in the `data` folder. Before you run our executables you have to add the renderer library to you `LD_LIBRARY_PATH`:

```
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:<your_base_folder>/render_lib
```

After this is done, change into the new working directory and execute:

```
> ../train_trees
```

This will train a random forest. Once training is finished, execute

```
> ../test_pose_estimation -t0 1 -nD
```

This will process all test images of the first object. Finally, it should report approximately 70% poses estimated correctly. Note that this setup is different than the setup in our paper, where we processed all 20 objects, added in-plane rotation, and depth features were allowed to learn context. If you want to reproduce this experiment use `create_split_full.py`, and add the following flags to `train_trees` and `test_pose_estimation`: `-amin -45 -amax 45 -as 15 -bgs 1`.

References

[1] Brachmann, E., Krull, A., Michel, F., Gumhold, S., Shotton, J., Rother, C.: Learning 6d object pose estimation using 3d object coordinates. In Fleet, D., Pajdla, T., Schiele, B., Tuytelaars, T., eds.: Computer Vision – ECCV 2014. Volume 8690 of Lecture Notes in Computer Science. Springer International Publishing (2014) 536–551

Table 1: Command line arguments for `train_trees`

Command Line Flag	Std. Val.	Description
-tc <val>	3	Tree count: Determines how many trees to train for the forest.
-fc <val>	1000	Feature count: Determines the size of the feature pool generated at each tree depth layer.
-wd <val>	1	Depth feature weight: Determines the ratio of depth features in the feature pool.
-wc <val>	1	Color feature weight: Determines the ration of color features in the feature pool.
-g <val>	20	Color noise: Adds Gaussian noise with the specified standard deviation to the color channels of the training images.
-mo <val>	20	Maximum offset: Determines the maximum size of color resp. depth features in the feature pool. It is measured in pixel meters.
-tp <val>	1000	Training pixels: Determines the number of samples drawn from each training image when learning the structure of the trees.
-tfr <val>	5	Training pixel factor for regression: Determines the number of samples drawn from each training image when learning the leaf distributions. The number of samples is $< tfr > \times < tp >$.
-ms <val>	50	Minimum samples: A node is only split further if at least this many samples arrive at this node (stopping criterion).
-md <val>	64	Maximum depth: Limits the maximum depth of a tree (stopping criterion).
-bgs <val>	0	Background strategy: If set to 1, depth features are allowed to look at the depth map outside the object mask during training. This way, they can learn object context.
-amin <val>	0	Minimum in-plane angle: -amin, -amax and -as allow you to add in-plane rotation to your training images. Each training image is rotated from -amin to -amax in -as steps. This increases the amount of training images. In the standard setting, no in-plane rotation is added.
-amax <val>	0	Maximum in-plane angle: See -amin.
-as <val>	1	In-plane angle step: See -amin.

Table 2: Command line arguments for `test_pose_estimation`. Note that all training parameters are valid, too.

Command Line Flag	Std. Val.	Description
-tO <val>	1	Test object: ID of the object you want to test. The ID is determined by the alphabetical order of the object data folders in `training`.
-rO	-	Rotation object: Attach this flag if the object specified is rotational symmetric. It will change the calculation of the pose error.
-nD	-	No display: Switches to batch mode. It will process all test images of the object specified non-stop.
-rI <val>	210	RANSAC iterations: Numbers of hypothesis drawn for each test image.
-rB <val>	25	Refine best: Number of best hypothesis that undergo local refinement.
-rRI <val>	100	RANSAC refinement iterations: Maximum number of local refinement steps done for each hypothesis.